Interactive Formal Verification (L21)
Exercises and Marking Scheme

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Michaelmas Term, 2021

Interactive Formal Verification consists of twelve lectures and four practical sessions. The handouts for the first two practical sessions will not be assessed. You may find that these handouts contain more work than you can complete in an hour, but you are not required to complete them: they are merely intended to be instructive. Many more exercises can be found at http://isabelle.in.tum.de/exercises/, but they tend to be easy. The assessed exercises are considerably harder, as you can see by looking at those of previous years.

The handouts for the last two practical sessions determine your final mark (50% each). For each assessed exercise, please complete the indicated tasks and write a brief document explaining your work. You may earn additional credit by preparing this document using Isabelle's theory presentation facility.[1] Alternatively, write the document using your favourite word processing package. Please ensure that your specifications are correct (because proofs based on incorrect specifications could be worthless) and that your Isabelle theory actually runs.

Each assessed exercise is worth 100 marks.

- 50 marks are for completing the tasks. Proofs should be competently done and tidily presented. Be sure to delete obsolete material from failed proof attempts. Excessive length (within reason) is not penalised, but slow or redundant proof steps may be. Sledgehammer may be used, but multi-line sledgehammer proofs can be unreadable and should not be presented in their raw form. Avoid inserting **apply** commands before the **proof** keyword.

- 20 marks are for a clear, basic write-up. It can be just a few pages, and probably no longer than 6 pages. It should explain your proofs, preferably displaying these proofs if they are not too long. It could

[1] See section 4.2 of the Isabelle/HOL Tutorial, https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2021/doc/tutorial.pdf.

perhaps outline the strategic decisions that affected the shape of your proof and include notes about your experience in completing it. Please don't copy the text of the exercises into your own write-up.

- The final 30 marks are for exceptional work. To earn some of these marks, you may need to vary your proof style, maybe expanding some **apply**-style proofs into structured proofs. The point is not to make your proofs longer (brevity is a virtue) but to demonstrate a variety of Isabelle skills, perhaps even techniques not covered in the course. Taking the effort to make your proofs more readable can help. Even better, strive for proofs that are direct and insightful; untidy or circuitous proofs and needless complexity can lose marks.

  An exceptional write-up also gains a few marks in this category. Very few students will gain more than half of these marks, but note that 85% is a very high score.

Isabelle theory files for all four sessions can be downloaded from the course materials website. These files contain necessary Isabelle declarations that you can use as a basis for your own work.

You must work on these assignments as an individual; collaboration is forbidden. Copying material found elsewhere counts as plagiarism. Here are the deadline dates. Exercises are due at 12 NOON.

- 1st exercise: Wednesday, 3 November 2021
- 2nd exercise: Wednesday, 17 November 2021

For each exercise, submit both the Isabelle theory file and the accompanying write-up by the deadline, using Moodle.

1 Replace, Reverse and Delete

Define a function `replace`, such that `replace x y zs` yields `zs` with every occurrence of `x` replaced by `y`.

```
consts replace :: "'a ⇒ 'a ⇒ 'a list ⇒ 'a list"
```

Prove or disprove (by counterexample) the following theorems. You may have to prove some lemmas first.

```
theorem "rev(replace x y zs) = replace x y (rev zs)"
theorem "replace x y (replace u v zs) = replace u v (replace x y zs)"
theorem "replace y z (replace x y zs) = replace x z zs"
```

Define two functions for removing elements from a list: `del1 x xs` deletes the first occurrence (from the left) of `x` in `xs`, `delall x xs` all of them.

```
consts del1   :: "'a ⇒ 'a list ⇒ 'a list"
       delall :: "'a ⇒ 'a list ⇒ 'a list"
```

Prove or disprove (by counterexample) the following theorems.

```
theorem "del1 x (delall x xs) = delall x xs"
theorem "delall x (delall x xs) = delall x xs"
theorem "delall x (del1 x xs) = delall x xs"
theorem "del1 x (del1 y zs) = del1 y (del1 x zs)"
theorem "delall x (del1 y zs) = del1 y (delall x zs)"
theorem "delall x (delall y zs) = delall y (delall x zs)"
theorem "del1 y (replace x y xs) = del1 x xs"
theorem "delall y (replace x y xs) = delall x xs"
theorem "replace x y (delall x zs) = delall x zs"
theorem "replace x y (delall z zs) = delall z (replace x y zs)"
theorem "rev(del1 x xs) = del1 x (rev xs)"
theorem "rev(delall x xs) = delall x (rev xs)"
```

2 Power, Sum

2.1 Power

Define a primitive recursive function *pow x n* that computes x^n on natural numbers.

consts
```
  pow :: "nat => nat => nat"
```

Prove the well known equation $x^{m \cdot n} = (x^m)^n$:

theorem `pow_mult: "pow x (m * n) = pow (pow x m) n"`

Hint: prove a suitable lemma first. If you need to appeal to associativity and commutativity of multiplication: the corresponding simplification rules are named `mult_ac`.

2.2 Summation

Define a (primitive recursive) function *sum ns* that sums a list of natural numbers: $sum[n_1, \ldots, n_k] = n_1 + \cdots + n_k$.

consts
```
  sum :: "nat list => nat"
```

Show that *sum* is compatible with *rev*. You may need a lemma.

theorem `sum_rev: "sum (rev ns) = sum ns"`

Define a function *Sum f k* that sums *f* from 0 up to $k-1$: $Sum\ f\ k = f\ 0 + \cdots + f(k-1)$.

consts
```
  Sum :: "(nat => nat) => nat => nat"
```

Show the following equations for the pointwise summation of functions. Determine first what the expression `whatever` should be.

theorem `"Sum (%i. f i + g i) k = Sum f k + Sum g k"`
theorem `"Sum f (k + l) = Sum f k + Sum whatever l"`

What is the relationship between `powSum_ex.sum` and `Sum`? Prove the following equation, suitably instantiated.

theorem `"Sum f k = sum whatever"`

Hint: familiarize yourself with the predefined functions `map` and `[i..<j]` on lists in theory List.

3 Assessed Exercise I: Baby Theory of Clauses

This exercise concerns propositional satisfiability checking: clause form and the elements of the DPLL procedure. A *clause* is conceptually a disjunction of *literals*, each of which is a (possibly negated) propositional variable. Clauses are represented by finite maps from variables to their polarity, `True` or `False`, where `False` indicates a negated variable. Variables are simply natural numbers.

We import the theory of finite maps, `HOL-Library.Finite_Map`.

```
type_synonym var = nat
type_synonym clause = "var ⇀ bool"
```

Next, we need a function to return the set of variables present in a set of clauses. A set of clauses is represented here by a list.

```
vars :: "clause list ⇒ var set"
```

Task 1 *Define the function* `vars` *above. (*Hint*: you will need the function* `dom`*.) Then prove the following consequences of your definition. [5 marks]*

```
lemma vars_Nil: "vars [] = {}"

lemma vars_Cons: "vars (cl # cls) = dom cl ∪ vars cls"
```

To instantiate the variable `v` with the boolean value `b`, a clause where `v` appears with polarity `b` must be deleted (for example, where `v` is positive and is being instantiated to `True`). Otherwise we delete `v` from the clause and continue recursively to the other clauses. (This process is called unit propagation.)

```
fun inst :: "var ⇒ bool ⇒ clause list ⇒ clause list"
  where "inst v b [] = []"
  |     "inst v b (cl#cls) =
           (if cl v = Some b then inst v b cls
            else cl(v:=None) # inst v b cls)"
```

Task 2 *Prove the following two facts about the definition above. [5 marks]*

```
lemma vars_inst: "vars (inst v b cl) ⊆ vars cl - {v}"

lemma inst_trivial: "v ∉ vars A ⟹ inst v b A = A"
```

The semantics of a set of clauses is defined in terms of environments, which are simply functions from variables to values, `Clauses_ex.var ⇒ bool`. Given an environment `e`, a clause is true if one of its literals evaluates to true in `e`:

definition `"evalC e cl ≡ ∃v ∈ dom cl. cl v = Some (e v)"`

We can easily extend `evalC` to sets of clauses by defining

```
eval :: "(var ⇒ bool) ⇒ clause list ⇒ bool"
```

Task 3 *Complete the definition of* `eval` *above. Then prove the following consequences of these definitions. Note that the last two relate to the treatment of unit clauses and case splits in the DPLL procedure.* *[8 marks]*

lemma `eval_Nil: "eval e []"`

lemma `eval_Cons: "eval e (cl#cls) ⟷ evalC e cl ∧ eval e cls"`

lemma `eval_notin_vars:`
  assumes `"eval e cls" "v ∉ vars cls"`
  shows `"eval (e (v:= b)) cls"`

lemma `unit_clause:`
  assumes `"eval e cls" "cl ∈ set cls" "dom cl = {v}"`
  shows `"cl v = Some (e v)"`

lemma `eval_cases:`
  assumes `"eval (e (v:= True)) cls" "eval (e (v:= False)) cls"`
  shows `"eval e cls"`

Task 4 *Prove the following lemma, which is about the semantics of an instantiated set of clauses.* *[12 marks]*

lemma `eval_inst: "eval e (inst v b cls) = eval (e (v:= b)) cls"`

Pure literals are those that appear with one polarity exclusively through the entire set of clauses. We first define a function to return the set of variables that appear with a given polarity `b`. We then define pure literals, using set difference to express exclusively,.

definition `signed_vars :: "bool ⇒ clause list ⇒ var set"`
  where `"signed_vars b ≡ Union ∘ set ∘ (map (λm.{a. m a = Some b}))"`

definition `"pure b cls ≡ signed_vars b cls - signed_vars (¬b) cls"`

Task 5 *Prove the following lemma, stating that a pure literal* `v` *of polarity* `b` *can be assumed to be set true in any model of a set of clauses.* *[20 marks]*

lemma `pure_literal_removal:`
  assumes `"eval e cls" "v ∈ pure b cls"`
  shows `"eval (e (v:= b)) cls"`

No proof should require more than 25 lines, but be careful in your choice of induction rules.

4 Assessed Exercise II: Prime Power Divisors

This exercise concerns the maximum power of a prime that divides a natural number. It imports `HOL-Computational_Algebra.Primes`, the theory of prime numbers.

First, we establish that the maximum power of a divisor exists.

Task 1 *Prove the following three lemmas. The type constraint* `nat` *is essential here. (Why?) The variable* `p` *will typically be prime, but the weaker condition* `2 ≤ p` *is often adequate.* *[5 marks]*

```
lemma power_dvd_nonempty:
  fixes n::nat shows "{j. p^j dvd n} ≠ {}"

lemma power_Max_dvd:
  fixes n::nat
  assumes "n > 0" "2 ≤ p"
  shows "p ^ Max{k. p ^ k dvd n} dvd n"

lemma Max_power_dvd_ge:
  fixes n::nat
  assumes "p ^ l dvd n" "n > 0" "2 ≤ p"
  shows  "l ≤ Max{k. p ^ k dvd n}"
```

We extend the definiton of this maximum power to return 0 in the undefined cases:

```
definition index where
   "index p n ≡ if p ≤ Suc 0 ∨ n = 0 then 0 else Max {j. p^j dvd n}"
```

Task 2 *Prove the following, which justifies the definition above.* *[5 marks]*

```
lemma pow_divides_index:
   "p^j dvd n ⟷ n = 0 ∨ p = Suc 0 ∨ j ≤ index p n"
```

Task 3 *Prove the following obvious-looking result, which turns out to be surprisingly difficult. Hint: to reason about* `Max`(A) *where* A *is a complicated set, simplify* A *using a previously-proved, carefully chosen set identities.* *[15 marks]*

```
lemma index_step:
  "index p (p*n) = (if p ≤ Suc 0 ∨ n=0 then 0 else Suc (index p n))"
```

A problem with `Max` is that it is not computational. But we could instead define the index function using plain recursion.

```
fun index_rec where
  "index_rec p n = (if p ≤ Suc 0 ∨ n = 0 ∨ ¬ p dvd n then 0
                    else Suc (index_rec p (n div p)))"
```

With this version, we can evaluate expressions and use **quickcheck** to check lemma statements.

```
value "index_rec 3 2430"
```

Task 4 *Prove that the functions* `index` *and* `index_rec` *really are equivalent, as formulated below.* *[10 marks]*

```
lemma index_index_rec: "index p n = index_rec p n"
```

Task 5 *The following definition, which refers to the cardinality of the set of powers, is also equivalent to* `index`*. Prove it.* *[15 marks]*

```
lemma index_card_def:
  "index p n = (if p ≤ Suc 0 ∨ n = 0 then 0
                else card {j. 1 ≤ j ∧ p^j dvd n})"
```

No proof needs to be longer than 35 lines. If yours is turning into a monster, consider whether to start again.